

# Field Programmable Gate Arrays (FPGAs) for Enhancing the Speed and Energy Efficiency of Quantum Dynamics Simulations

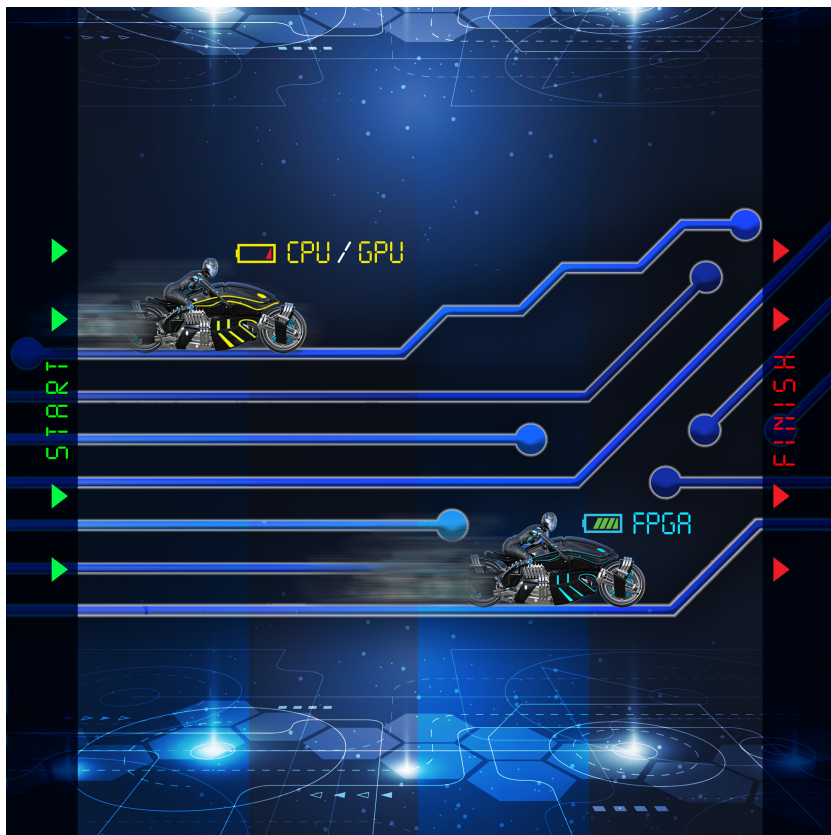
José M. Rodríguez-Borbón,<sup>†</sup> Amin Kalantar,<sup>†</sup> Sharma S. R. K. C. Yamijala,<sup>‡</sup> M.  
Belén Oviedo,<sup>¶</sup> Walid Najjar,<sup>†</sup> and Bryan M. Wong<sup>\*,‡</sup>

<sup>†</sup>*Department of Computer Science & Engineering, University of California-Riverside,  
Riverside, CA 92521, USA*

<sup>‡</sup>*Department of Chemical & Environmental Engineering, Materials Science & Engineering,  
Department of Chemistry, and Department of Physics & Astronomy, University of  
California-Riverside, Riverside, CA 92521, USA*

<sup>¶</sup>*Departamento de Química Teórica y Computacional, Facultad de Ciencias Químicas,  
Universidad Nacional de Córdoba, Instituto de Investigaciones en Fisicoquímica de Córdoba  
(INFIQC), UNC-CONICET, Córdoba X5000HUA, Argentina*

E-mail: [bryan.wong@ucr.edu](mailto:bryan.wong@ucr.edu), <http://www.bmwong-group.com>



TOC Graphic

### Abstract

We present the first application of field programmable gate arrays (FPGAs) as new, *customizable* hardware architectures that can be harnessed for the fast and energy-efficient calculation of quantum dynamics simulations of large chemical/material systems. Instead of tailoring the software to fixed hardware (which is the typical case for writing quantum chemistry code for CPUs/GPUs), FPGAs allow us to *directly customize the underlying hardware* – even at the level of specific electrical signals in the circuit – to give a truly optimized computational performance for complex quantum dynamics calculations. By offloading the most intensive and repetitive calculations onto an FPGA, we show that the computational performance of our hardware implementation for real-time electron dynamics calculations can even exceed that of optimized commercial mathematical libraries running on high-performance GPUs. In addition to

this impressive computational speedup, we show that FPGAs are immensely energy-efficient and consume 4 times less energy than modern GPU or CPU architectures. These energy savings are a practical and important metric for supercomputing centers (several of which exceed over \$1 million in power costs alone), as exascale computing capabilities become more widespread and commonplace. Taken together, the implementation techniques and performance metrics of our study demonstrate that FPGAs could play a promising role in upcoming quantum chemistry and materials science applications, particularly for the acceleration and energy-efficient execution of quantum dynamics calculations.

## 1. Introduction

Modern quantum chemistry techniques depend critically on massively parallelized computational hardware to enable accurate calculations of the many-body electronic Schrödinger equation. Indeed, over the past two decades, the quantum chemistry community has witnessed tremendous technological advancements in computing that have enabled simulations of chemical/material systems of increasing complexity. These advancements have become even more prominent as we rapidly approach the dawn of exascale computing, with machines capable of performing a million trillion floating-point calculations per second.<sup>1-3</sup> However, to enable these massive calculations, recent exascale computing guidelines<sup>4-6</sup> have strongly cautioned that this increase in computing power should only require a modest increase in power consumption (to offset both operation costs and deleterious climate change effects). Maintaining this delicate balance between computational performance vs. energy efficiency is extremely difficult since recent reports<sup>7,8</sup> have shown that even small supercomputing centers regularly consume 500-1000 kW of power over the course of the year, resulting in over \$1 million for power costs alone. These estimates do not even account for cooling costs, which have been reported to make up 25–50% of total power required by large data centers.<sup>9</sup> To partially mitigate these issues, this work is a first attempt to address these emerging

parallelization and power-usage concerns via our use of new computational hardware, known as Field Programmable Gate Arrays (FPGAs), for quantum chemical calculations.

FPGAs are re-configurable hardware architectures that are comprised of an array of millions of connection blocks (CBs) and configurable logic blocks (CLBs) connected by metal channels that can be configured to create any desired circuit (cf. Fig. 1). Like GPUs, FP-

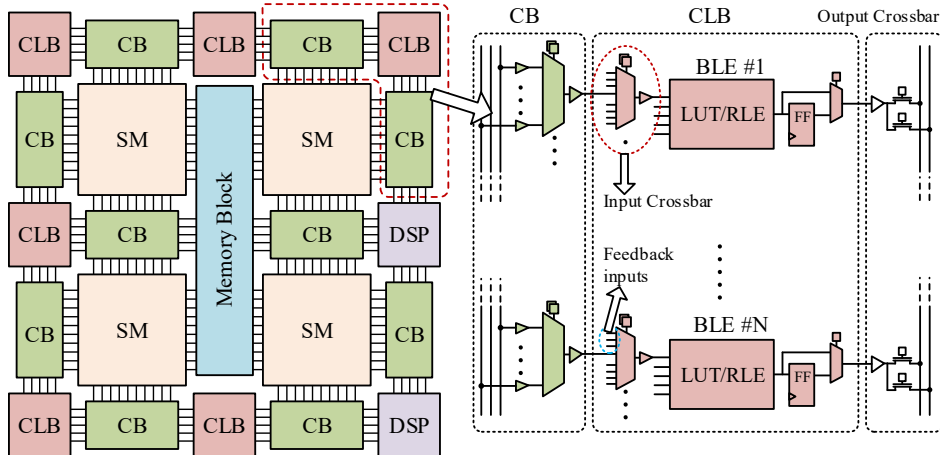


Figure 1: (left) Hardware schematic of a typical (island-style) FPGA architecture where connection blocks (CBs) and configurable logic blocks (CLBs) are seamlessly integrated with switch matrices (SMs) and physical wires. (right) Detailed hardware schematic within the CB/CLB. Each CLB is comprised of a cluster of basic logic elements (BLEs) which themselves are composed of re-configurable logic elements (RLEs) known as look up tables (LUTs) and a storage element known as a flip flop (FF).

GAs are ideal for parallelization, but with the added advantage of having re-programmable circuitry *at the hardware level* to enable even further parallelization and significant energy gains. Instead of tailoring the software to fixed hardware (as is done in CPUs/GPUs), *FP-GAs follow a different philosophy and allow the customization of hardware that best meets the specific computational application under study*. This configurability is possible since FPGAs are programmed in a hardware description language (HDL) that enables ultimate control by routing individual electrical signals through user-defined gates directly in the FPGA hardware. Although the use of a hardware description language may seem impractical, this programming structure allows a user to carry out several different mathematical instructions at the same time on a single FPGA, resulting in a truly optimized computational

performance.

While our use of re-programmable hardware bears some resemblance to the techniques used by Shaw and co-workers to accelerate molecular dynamics calculations with the customized Anton machine,<sup>10–12</sup> the approach utilized in our work has several distinct differences. In particular, Anton belongs to a class of computing architectures known as application-specific integrated circuits (ASICs), which are extremely expensive to design and hard to modify when new types of calculations are desired. Compared to ASICs, FPGAs can be re-configured *at the hardware level* for a variety of applications, while also being significantly less expensive to manufacture or power. As a tangible example of these advantages, FPGAs are currently used by Microsoft to accelerate the Bing search engine, and the FPGA market is expected to increase to 7.3 billion in 2022.<sup>13</sup> As such, the FPGA-based approach used in this work is also a viable solution for parallelizing calculations at the hardware level without requiring a prohibitively expensive infrastructure.

In this work, we present the first application of FPGAs for use in massively parallelized quantum dynamics of large chemical systems (up to 3,338 atoms). To this end, we have modified our real-time, time-dependent density functional tight binding (RT-TDDFTB) code to make use of the highly parallelized architecture of FPGAs. Our motivation for implementing RT-TDDFTB with FPGAs is two-fold: (1) the RT-TDDFTB formalism scales favorably with system size and allows us to compare the performance of FPGAs with other parallelized hardware approaches, namely the GPU-enabled DFTB and RT-TDDFTB implementations used in our prior work,<sup>14–18</sup> and (2) the techniques presented in this work can be used as a first step towards full DFT-based electron dynamics or other DFTB-based methodological developments such as large-scale non-adiabatic dynamics calculations. Since FPGAs have not been previously used by the quantum dynamics community, we give a detailed description of our approach (at both a hardware and programming level of detail) in conjunction with several benchmark tests. Finally, we present comparisons of the speed and energy gains obtained with FPGAs compared to CPUs/GPUs to highlight their immense promise for

carrying out massively parallelized quantum dynamics calculations on these novel hardware architectures.

## 2. Theory and Computational Methodology

Before proceeding with a detailed description of our FPGA parallelization enhancements, we first give a brief overview of the RT-TDDFTB formalism. Over the past few years, the RT-TDDFTB approach has garnered significant attention as an extremely efficient technique for probing the non-equilibrium electron dynamics of extremely large chemical systems. Specifically, we and others have used the RT-TDDFTB approach to understand photo-injection dynamics in dye-sensitized TiO<sub>2</sub> solar cells,<sup>19–21</sup> many-body interactions in solvated nanodroplets,<sup>17</sup> and excitation energy transfer dynamics in plasmonic arrays.<sup>15,16</sup> These real-time quantum dynamics calculations are carried out by applying a time-dependent electric field to the initial ground state density matrix, resulting in an explicitly time-dependent Hamiltonian  $\hat{\mathbf{H}}(t)=\hat{\mathbf{H}}^0 - \mathbf{E}_0(t) \cdot \hat{\boldsymbol{\mu}}(t)$ , where  $\mathbf{E}_0(t)$  is the applied electric field, and  $\hat{\boldsymbol{\mu}}(t)$  is the dipole moment operator. Since the quantum system is directly propagated in the time-domain,  $\mathbf{E}_0(t)$  can have any arbitrary time-dependent form. For example, if  $\mathbf{E}_0(t)$  is a Dirac delta function,  $\mathbf{E}_0(t) = \delta(t - t_0)$ , this yields an optical absorption spectrum (obtained after a Fourier transform of the time-evolving dipole moment). However, if  $\mathbf{E}_0(t)$  takes the form of a sinusoidal perturbation, it represents a continuous interaction of the system with monochromatic light in the time domain. When either of these time-dependent fields are applied, the density matrix  $\hat{\boldsymbol{\rho}}$  evolves according to the Liouville-von Neumann equation of motion which, in the nonorthogonal-DFTB basis, is given by

$$\frac{\partial \hat{\boldsymbol{\rho}}}{\partial t} = \frac{1}{i\hbar}(\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}] \cdot \hat{\boldsymbol{\rho}} - \hat{\boldsymbol{\rho}} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}] \cdot \mathbf{S}^{-1}), \quad (1)$$

where  $\hat{\mathbf{H}}$  is the Hamiltonian matrix (which implicitly depends on the density matrix),  $\mathbf{S}^{-1}$  is the inverse of the overlap matrix, and  $\hbar$  is Planck’s constant. When the applied incident

fields are smaller than the internal fields in a molecule or material, the system is in the linear response regime.<sup>22</sup> Under these conditions, the time evolution of the dipole moment operator can be expressed as the convolution between the applied electric field perturbation, resulting in the following response function of the system

$$\langle \hat{\boldsymbol{\mu}}(t) \rangle = \int_0^\infty \boldsymbol{\alpha}(t - \tau) \mathbf{E}(\tau) d\tau, \quad (2)$$

where  $\mathbf{E}(\tau)$  is the electric field that induces a perturbation in the Hamiltonian, and  $\boldsymbol{\alpha}(t - \tau)$  is the polarizability tensor. Upon application of the convolution theorem, Equation 2 can be expressed in the frequency domain as  $\langle \hat{\boldsymbol{\mu}}(\omega) \rangle = \boldsymbol{\alpha}(\omega) \mathbf{E}(\omega)$ . The imaginary part of the average polarizability,  $\bar{\alpha}$  is an experimentally measurable quantity related to the photoabsorption cross section by the expression  $\sigma(\omega) = 4\pi\omega/c \cdot \text{Im}(\bar{\alpha})$ , where  $c$  is the speed of light, and  $\text{Im}(\bar{\alpha})$  is the imaginary part of the average polarizability. In this work, we utilized the DFTB+ code<sup>23</sup> to construct the ground-state Hamiltonian, overlap matrix elements, and the initial single-electron density matrix within the self-consistent DFTB approach. With these ground-state quantities pre-computed, excited-state electron dynamics calculations were carried out with a customized RT-TDDFTB implementation on both GPU and FPGA hardware architectures, as described in the following paragraphs.

**FPGA Hardware.** Fig. 2 depicts a high-level schematic of the FPGA hardware used in this work, which is composed of an Intel Xeon CPU E5-2460 interfaced with a Virtex-7 FPGA.<sup>24</sup> This specific FPGA configuration has 32 memory channels, each of which has a theoretical bandwidth of 1.33 GB/s. As such, to achieve maximum I/O performance, we configured our FPGA to execute read/write requests of 64 bytes aligned to 64-byte addresses (the FPGA can carry out read/write requests of 8, 4, 2, or 1 byte, but these smaller sizes result in a lower I/O performance). Our FPGA implementation was written using Verilog HDL, and our hardware design was simulated with ModelSim<sup>25</sup> to test its accuracy. Our FPGA implementation was synthesized, placed, and routed with Vivado 17.3.<sup>26</sup> The target

frequency of our FPGA was set to 167 MHz. Our implementation was modified to accept real- and complex-valued single precision floating point matrices as input (discussed further in Sections 4 and 5). All arithmetic operations were implemented on Xilinx cores<sup>27</sup> by taking advantage of either digital signal processors (DSP) or lookup tables (LUTs).

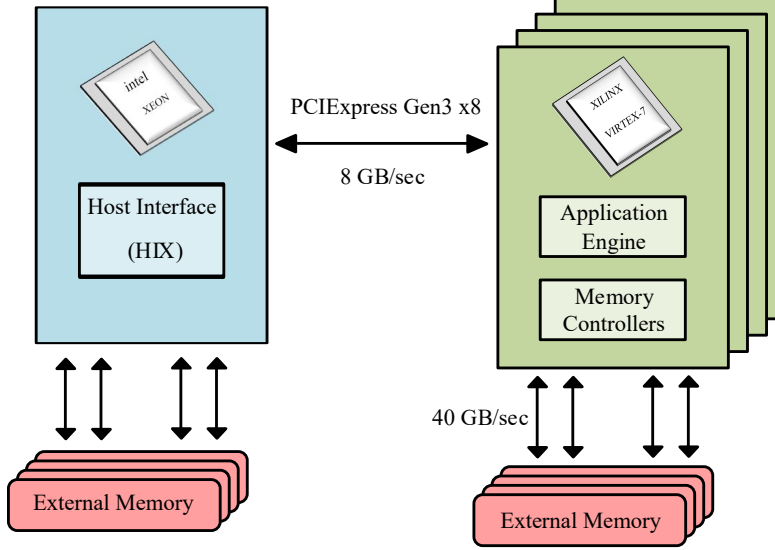


Figure 2: Schematic of the Micron Wolverine FPGA used in this work. This hardware architecture is comprised of a CPU with one FPGA attached via a PCIeExpress Line. The FPGA is first configured with the specific simulation to be executed, and the CPU sends commands to the FPGA via the host interface. These commands include operations such as writing (reading) data to (from) the FPGA external memory, executing the computation, and querying the status of the computation.

**GPU and CPU Hardware Comparisons.** To assess the performance of our FPGA implementation against other computational hardware, we also examined computational timings and energy expenditures of both GPUs and CPUs. For our GPU benchmark tests, we utilized an NVIDIA K40 GPU (we used only one of the K40 devices within a K80 GPU) equipped with an Intel Xeon E5-520 processor and 24 GB of RAM. To ensure a fair assessment of computational efficiency, our GPU-based RT-TDDFTB code was compiled with CUDA (release 9.0) in conjunction with the CUBLAS linear algebra library<sup>28</sup> to achieve optimal computational performance on the GPU. In all our GPU-based tests/comparisons, error correction capabilities (ECC) were disabled. For our CPU tests, we utilized an Intel

Xeon E5-2643 V3 processor operating at 3.40 GHz with 256 GB of RAM. Similar to our GPU benchmark tests, our CPU implementation utilized optimized routines within the Intel Math Kernel Library (MKL) in conjunction with OpenMP for multi-threading.

To enhance the efficiency of our RT-TDDFTB calculations, the majority (roughly 70%) of the computation of Equation 1 was offloaded to a co-processor (either an FPGA or a GPU as described previously, and additional implementation details are given in Section 4). To enable this efficiency, Equation 1 was computed in multiple steps as follows.

- 1) In the CPU, the self-consistent charge (SCC) and non-SCC Hamiltonian matrices are parsed in conjunction with the overlap matrix, orbital-wise electron fillings, and spatial coordinates of the system. The corresponding data structures for the density, overlap, and Hamiltonian matrices are subsequently generated.
- 2) Within the CPU, the matrix product  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]$  is computed.
- 3) The matrices  $\hat{\boldsymbol{\rho}}(t)$ ,  $\hat{\boldsymbol{\rho}}(t)^T$ ,  $\hat{\boldsymbol{\rho}}(t - \Delta t)$ , and the matrix resulting from step 2 are transferred to the co-processor (i.e., a GPU or an FPGA).
- 4)  $\hat{\boldsymbol{\rho}}_1(t + \Delta t) = \frac{1}{i\hbar} \{(\mathbf{S}^{-1} \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]) \hat{\boldsymbol{\rho}}(t)\} (2\Delta t) + \hat{\boldsymbol{\rho}}(t - \Delta t)$  is computed in the co-processor.
- 5)  $\hat{\boldsymbol{\rho}}_2(t + \Delta t) = \frac{1}{i\hbar} \{(\mathbf{S}^{-1} \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]) \hat{\boldsymbol{\rho}}(t)^T\} (2\Delta t)$  is computed in the co-processor.
- 6) The resulting matrices from steps 4 and 5 are transferred to the CPU where the three-point formula  $\hat{\boldsymbol{\rho}}(t) = \hat{\boldsymbol{\rho}}_1(t) - \hat{\boldsymbol{\rho}}_2(t)^T$  is computed (i.e., a simple subtraction of two pre-computed quantities with little computational overhead).
- 7) The density  $\hat{\boldsymbol{\rho}}$  and Hamiltonian  $\hat{\mathbf{H}}$  matrices are updated in the CPU.
- 8) The entire process starting with step 2 is repeated to propagate the electron dynamics for the desired time duration. The time-dependent charges, dipole moment, and density matrices are subsequently processed.

### 3. Chemical Systems and General FPGA Matrix Operations

Since the main focus of this work is to implement (which is a massive task in and of itself) and understand FPGA performance gains for computing electron dynamics, we have chosen a representative set of large chemical structures to assess its efficiency and computational scaling. To this end, we have constructed a set of hydrogen-terminated carbon nanoribbons<sup>29</sup> ranging from 62 – 3,338 atoms, and Fig. 3 depicts a subset of these structures as a function of size. It is worth mentioning that we specifically chose 3,338 atoms as our upper limit since this corresponds to the maximum matrix size that can be held in the memory of the GPU used in our performance benchmarks.

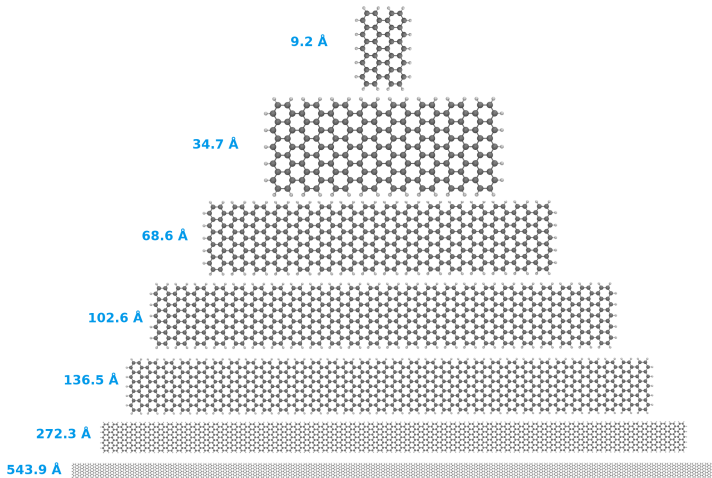


Figure 3: A representative subset of the carbon nanoribbons with various lengths examined in this work.

In computing the electron dynamics of these large nanoribbons, it is worth noting that both the Hamiltonian matrix  $\hat{\mathbf{H}}$  and the inverse of the overlap matrix  $\mathbf{S}^{-1}$  in Equation 1 are real-valued, whereas the density matrix  $\hat{\rho}$  is complex-valued. Moreover, while the density matrix  $\hat{\rho}(t)$  is dense, the matrix product  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\rho}]$  is sparse, which increases as a function of the nanoribbon size as shown in Fig. 4.

To efficiently parallelize our RT-TDDFTB calculations on FPGAs, we designed a soft-

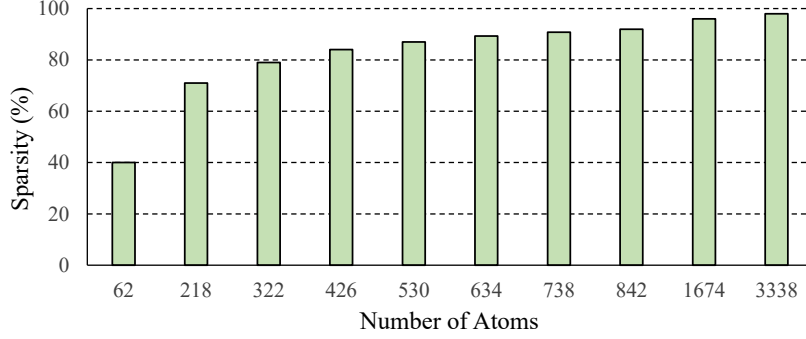


Figure 4: Sparsity of the matrix product  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}]$  as a function of nanoribbon size.

ware/hardware kernel that executes steps 4 – 5 (which are the most computationally demanding steps, as described in Section 2) and has the capacity for transferring matrices to and from the co-processor. Moreover, by supporting the matrix operations described in step 4, the implementation for the matrix operations described in step 5 is already satisfied since the addition of  $\hat{\boldsymbol{\rho}}(t - \Delta t)$  can be omitted in the latter step. To this end, we created a general-purpose hardware kernel to support the mathematical operation

$$\mathbf{C}^k = \alpha \mathbf{A} \mathbf{B} + \beta \mathbf{C}^{k-1}, \quad (3)$$

where the superscript  $\mathbf{k}$  denotes the  $k$ th iteration,  $\mathbf{A}$  is a real-valued matrix, and the matrices  $\mathbf{B}$  and  $\mathbf{C}$  along with the parameters  $\alpha$  and  $\beta$  are complex-valued. As such, the RT-TDDFTB simulations in the co-processor can be enabled by setting  $\mathbf{A} = \mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]$ ,  $\mathbf{B} = \hat{\boldsymbol{\rho}}(t)$  or  $\hat{\boldsymbol{\rho}}(t)^T$ , and  $\mathbf{C}^{k-1} = \hat{\boldsymbol{\rho}}(t - \Delta t)$ . In addition, the parameter  $\alpha$  was set to  $\frac{1}{i\hbar}(2\Delta t)$  while the parameter  $\beta$  is real and set to one. As described further in Sections 4 – 5, our kernel exploits the sparsity of  $\mathbf{A}$  and allows us to decrease both the input/output (I/O) and the computational complexity of the matrix operations to be offloaded to the FPGA.

## 4. Baseline FPGA Design and Architecture

Since FPGAs have not been previously used for quantum dynamics calculations, we first present a general (but detailed) hardware design for carrying out parallelized matrix multiplications. We designate this as our “baseline” FPGA hardware design, with Section 4.A. describing our baseline implementation for real-valued matrix multiplications and Section 4.B. giving our modifications for complex-valued matrix operations. Section 5 presents additional acceleration techniques tailored specifically to the efficient propagation of RT-TDDFTB electron dynamics on FPGAs.

### A. Real-Valued Matrix Multiplications on FPGAs

The multiplication of real-valued matrices on FPGAs continues to be a topic of interest,<sup>30–36</sup> and to enable the computations required by our RT-TDDFTB simulations (Eq. 3) we have modified a previous design<sup>31</sup> that was used for real-valued, dense matrix multiplication. We commence with Fig. 5, which depicts a general-purpose schematic for parallelization of real-valued matrix multiplication on FPGAs. To allow our baseline design to be completely general, the size of the matrices  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{C}$  are  $n \times m$ ,  $m \times l$ , and  $n \times l$ , respectively. Moreover, matrices  $\mathbf{A}$ ,  $\mathbf{B}$ , and  $\mathbf{C}$  have been partitioned into sub-blocks of size  $p \times m$ ,  $m \times p$ , and  $p \times p$ , respectively (with  $n/p$  and  $l/p$  being integer numbers). As described further in Section 6.B., the sizes of these sub-blocks were optimized to utilize most (approximately 70%) of the memory channels and block RAMs on our FPGA hardware platform. More specifically, the usage of both the I/O memory channels and block RAM was near its theoretical maximum (usage of resources beyond 70% in FPGAs can impair the placing and routing of the design), and further optimizations would lead to minimal/diminishing returns. Within this schematic, the computation of each block  $\mathbf{C}_{ij}$  can be obtained via multiplications of the corresponding blocks in  $\mathbf{A}$  and  $\mathbf{B}$  (i.e.,  $\mathbf{C}_{11} = \mathbf{A}_{11}\mathbf{B}_{11}$ ). More generally, each block  $\mathbf{C}_{ij}$  can be calculated as outer products ( $\cdot$ ) between the columns of block  $\mathbf{A}_{i1}$  and the rows of block

$\mathbf{B}_{1j}$  such that  $\mathbf{C}_{ij} = \mathbf{a}_1 \cdot \mathbf{b}_1 + \mathbf{a}_2 \cdot \mathbf{b}_2 + \dots + \mathbf{a}_m \cdot \mathbf{b}_m$ , where the column vector  $\mathbf{a}_i$  is the  $i$ th column of block  $\mathbf{A}_{i1}$  and the row vector  $\mathbf{b}_i$  is the  $i$ th row of block  $\mathbf{B}_{1j}$ . In the terminology of computational linear algebra algorithms, matrices having the form  $\mathbf{a}_k \cdot \mathbf{b}_k$  are rank-one matrices, and the addition of rank one matrices is called a rank one update.<sup>37,38</sup> By using these rank one updates, we can improve both I/O bandwidth and parallelism, since if one element of the column vector  $\mathbf{a}_k$  as well as the row vector  $\mathbf{b}_k$  are available, we can execute  $p$  multiply-and-accumulate operations simultaneously. Moreover,  $\mathbf{b}_k$  can be reused  $p$  times to improve the performance of matrix multiplications on FPGAs.<sup>30,31</sup>

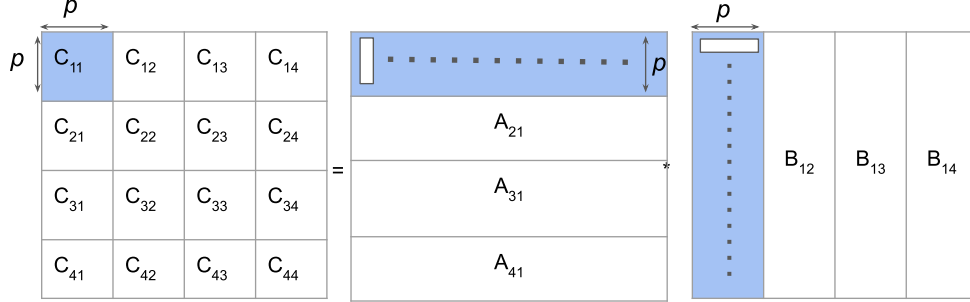


Figure 5: Schematic of parallelized matrix multiplication on FPGAs. The computation of the block  $\mathbf{C}_{11}$  can be obtained via the outer-products between the columns of  $\mathbf{A}_{11}$  and the rows of  $\mathbf{B}_{11}$ .

The general framework for carrying out these parallelized matrix multiplication operations on FPGAs is shown as a high-level flowchart in Fig. 6. This hardware engine is comprised of six modules designated as the *Scheduler*, *Reader*, *Read A Controller*, *Read B Controller*, *Multiply-and-Accumulate*, and the *Write C Controller*. All communication between these modules is executed via first-in first-out blocks (FIFOs).<sup>39</sup> While the design of FPGA *Readers*, *Writes*, and *Controllers* is generally well established, multiple designs have been proposed for the implementation of the *Multiply-and-Accumulate* unit. In this work, we have modified a previous design<sup>31</sup> in which one FIFO (shown in the top left of Fig. 7) stores the elements of the columns of matrix  $\mathbf{A}_{i1}$  (i.e., the column vectors  $\mathbf{a}_k$ ). Similarly, at the top of Fig. 7, the module has  $p$  FIFOs to store the rows of the matrix  $\mathbf{B}_{1j}$  (i.e.,

the row vectors  $\mathbf{b}_k$ ). In the center of Fig. 7,  $p$  *Multiply-and-Accumulate* units execute the multiply-and-accumulate operation. At the bottom of Fig. 7,  $p$  FIFOs are used for storing the final values of  $\mathbf{C}_{ij}$ . The leftmost FIFO stores the elements of the first column in  $\mathbf{C}_{ij}$ , and the FIFO next to it stores the elements of the second column in  $\mathbf{C}_{ij}$  and so on.

The dotted box to the right of Fig. 7 shows the components of a *Multiply-and-Accumulate* unit. A block RAM (BRAM) unit, which contains  $p$  addresses with 32 bits per address to hold single precision numbers, is used to store the partial values of  $\mathbf{C}_{ij}$ . A multiplexer is then used to multiplex one of the inputs of the adder. During most of the computation, the input to the adder is a numerical value from the BRAM; however, the multiplexer outputs zero when the calculation of a new block  $\mathbf{C}_{ij}$  starts.

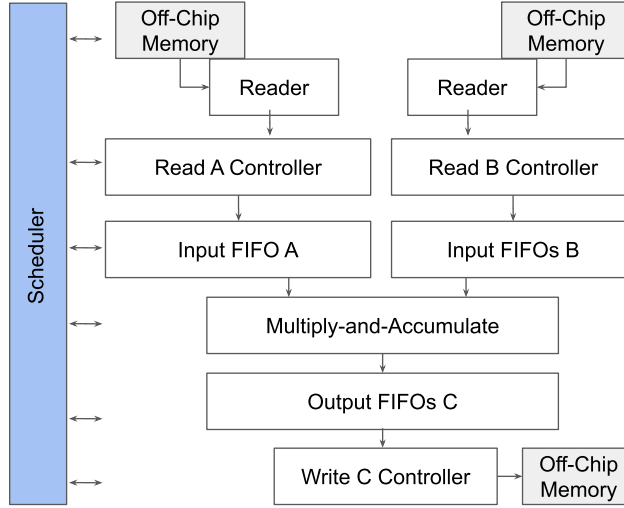


Figure 6: High-level view of the design for parallelizing our RT-TDDFTB simulations in FPGA hardware. In this figure, the *Scheduler* directs the execution of tasks to the other modules. The *Read (Write)* controller reads (writes) one input matrix from (to) the off-chip memory. Finally, the *Multiply-and-Accumulate* module executes the matrix multiplication operation.

The computation of  $\mathbf{C}_{ij}$  commences as follows. First, the *Controller* signals the reading of the first row of block  $\mathbf{B}_{1j}$  and the first column of block  $\mathbf{A}_{i1}$ , which are executed by the *Read B* and *Read A* Controller, respectively. These values are stored in the  $p$  FIFOs labeled  $b_{k,0}, \dots, b_{k,p-1}$  and the FIFO labeled  $a_{i,k}$  respectively. The *Controller* then commands the

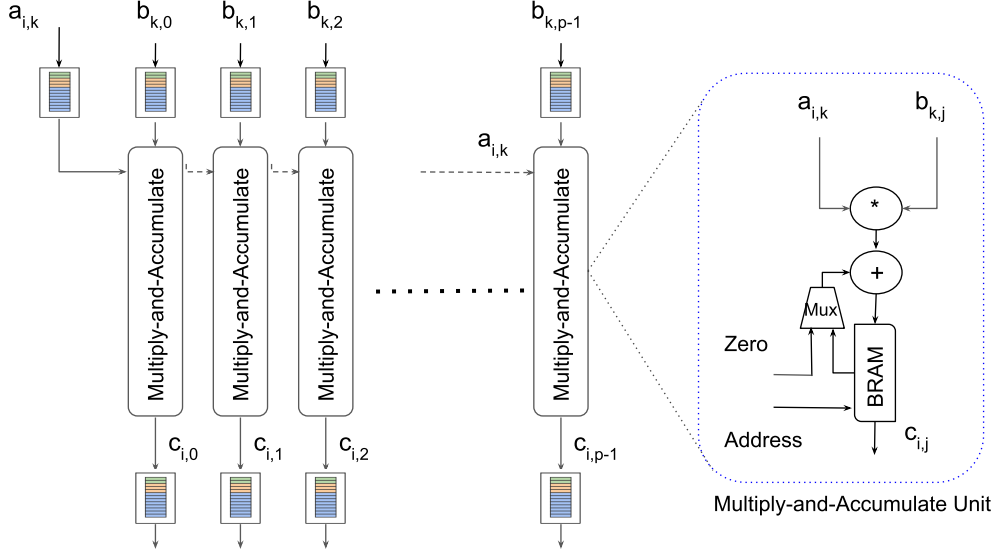


Figure 7: Hardware implementation of the *Multiply-and-Accumulate* module. This module executes the real-valued outer-products between the columns of matrix  $\mathbf{A}$  and the rows of matrix  $\mathbf{B}$ . The partial results are stored in block RAMs (BRAMs). The final results are stored in the FIFOs shown at the bottom. The components of the *Multiply-and-Accumulate* unit are shown in the dotted box on the right.

*Multiply-and-Accumulate* module to carry out  $p$  multiplications in parallel; i.e.,  $a[0, 0] * b[0, j]$  for  $j = 0, \dots, p - 1$ . The results of these multiplications are subsequently added to the zero values coming from the multiplexers and the results are stored in the BRAM at address zero. After the element  $a[1, 0]$  arrives to the top-left FIFO, the *Controller* signals the execution of  $p$  new multiplications and  $p$  new additions. Finally, the results are stored in RAM at address one, and this process continues until the outer product between the first column of  $\mathbf{A}_{i1}$  and the first row of  $\mathbf{B}_{1j}$  is completed.

In addition to the tasks mentioned above, the *Controller* directs the execution of the outer product between the second column of  $\mathbf{A}_{i1}$  and the second row of  $\mathbf{B}_{j1}$ . The results of these multiplications are then added to the previous values stored in the BRAMs. This process continues until the outer product between the last column of  $\mathbf{A}_{i1}$  and the last row of  $\mathbf{B}_{j1}$  is executed. At this point, the operation  $\mathbf{C}_{ij} = \mathbf{A}_{i1} \mathbf{B}_{1j}$  is completed, and the results are stored in the  $p$  BRAMs. The content of the BRAMs is written to the  $c_{i,0}, \dots, c_{i,p-1}$  FIFOs

one row at the time. Finally, the *Scheduler* signals to the *Write C Controller* to write the content of these FIFOs to the off-chip memory, and this process continues until all the  $\mathbf{C}_{ij}$  blocks are computed.

It is worth mentioning a few practical notes that can be used to enhance the efficiency of real-valued matrix multiplication on FPGAs (we implement some of these tactics in Section 6.D. on a new FPGA hardware architecture). First, the block  $\mathbf{C}_{ij}$  does not have to be square, and its size can be tailored to any specific FPGA hardware platform.<sup>30</sup> For example, if the block  $\mathbf{C}_{ij}$  has dimensions of  $p \times q$ , the parameter  $p$  can be increased to yield higher efficiency on FPGA platforms that have more on-chip memory. Second, for FPGAs with abundant floating point units (FPUs), the parameter  $q$  can be increased as well. Third, if the delay in the floating point addition is  $v$  cycles, it is desirable to have  $p \geq v$  to maintain computational efficiency. This constraint arises since the accumulations of the previous outer product must be finished before the next outer product starts, or the pipeline has to be stalled. Finally, if one has access to large FPGAs, or multiple FPGAs, several  $\mathbf{C}_{ij}$  blocks can be computed in parallel using the computational techniques discussed previously.

## B. Complex-Valued Matrix Multiplications on FPGAs

In this section, we describe our customized baseline design for complex-valued matrix multiplications on FPGAs. While the FPGA engine described in the previous section supports real-valued matrix multiplication in the expression  $\mathbf{C} = \mathbf{AB}$ , we implemented a new design for computing complex-valued matrix multiplications required for propagating RT-TDDFTB electron dynamics (cf. Eqs. 1 and 3). To support this new capability, we first compute an intermediate matrix  $\mathbf{T}_{ij}$  given by

$$\mathbf{T}_{ij} = \mathbf{A}_{i1}\mathbf{B}_{1j}, \quad (4)$$

where the blocks of the matrix  $\mathbf{A}$  and  $\mathbf{B}$  are real- and complex-valued, respectively. Fig. 8 depicts our customized complex-valued *Multiply-and-Accumulate* module that executes this

parallelized operation. Compared to the real-valued multiply-and-accumulate module shown previously in Fig. 7,  $p$  multiply-and-accumulate units have been added so as to compute the real ( $t_{i,k}^r$ ) and imaginary ( $t_{i,k}^i$ ) values of  $\mathbf{T}_{ij}$  in parallel. In this figure, the elements of  $\mathbf{T}_{ij}$  are serialized (one row at a time) into two FIFOs, labeled  $s^r$  and  $s^i$ , that contain the real and imaginary elements of  $\mathbf{T}_{ij}$ . With these values in hand, we next compute the following matrix

$$\mathbf{C}_{ij}^k = \alpha \mathbf{T}_{ij} + \beta \mathbf{C}_{ij}^{k-1}, \quad (5)$$

which is carried out by the *Complex Accumulator* module shown in Fig. 9. The design depicted in Fig. 9 has been harnessed with a new *Read C Controller* module. This additional FPGA module reads the complex values of matrix  $\mathbf{C}^{k-1}$  from the off-chip memory into the FIFOs labeled  $c^r$  and  $c^i$  (i.e., the real and imaginary parts of  $\mathbf{C}^{k-1}$ ). The *Complex Accumulator* module executes eight multiplications and six additions, with the real and imaginary values of  $\mathbf{C}_{ij}^k$  written into the FIFOs labeled  $t^r$  and  $t^i$ , respectively. At the end of the computation, the *Writer C Controller* writes the block  $\mathbf{C}_{ij}^k$  into the off-chip memory.

## 5. Optimized FPGA Design for Efficient Propagation of RT-TDDFTB Electron Dynamics

Having described our baseline FPGA design, we now present further optimizations that were added to speed up our RT-TDDFTB quantum dynamics calculations. In these simulations, as mentioned in Section 3, the matrix  $\mathbf{A} = \mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\boldsymbol{\rho}}(t)]$  is sparse, whereas the matrices  $\mathbf{B} = \hat{\boldsymbol{\rho}}(t)$  and  $\mathbf{C}^{k-1} = \hat{\boldsymbol{\rho}}(t - \Delta t)$  are dense. As a result, our baseline design was modified to take advantage of the sparsity of  $\mathbf{A}$  to satisfy the following three constraints:

- 1) Since the input matrix  $\mathbf{C}^{k-1}$  is dense, all the elements of the matrix  $\mathbf{T}_{ij}$  are required to execute the addition shown in Equation 5. Thus, a number of operations in the

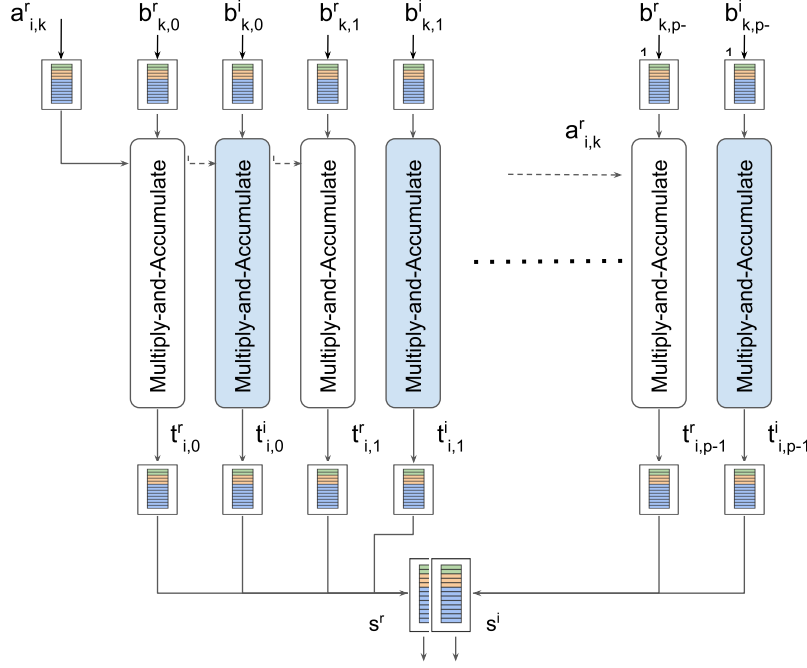


Figure 8: Hardware implementation of the *Complex Multiply-and-Accumulate* module. This module executes the complex outer-products between the real columns of matrix  $\mathbf{A}$  and the complex rows of matrix  $\mathbf{B}$ . The values of the resulting matrix are serialized to the  $s^r$  and  $s^i$  FIFOs at the bottom.

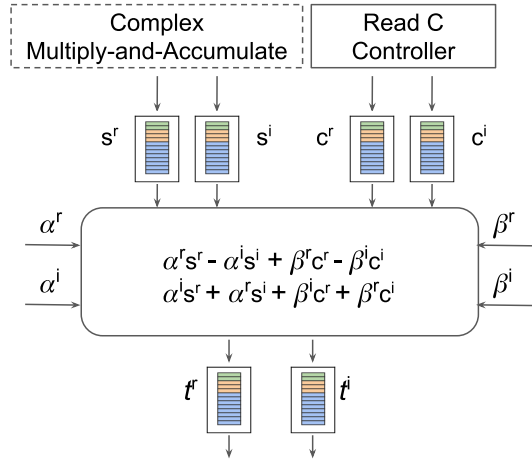


Figure 9: Hardware implementation of the *Complex Accumulator* module. This module executes the operation  $\alpha \mathbf{T}_{ij} + \beta \mathbf{C}_{ij}^{k-1}$ . The values of  $\mathbf{T}_{ij}$  are in the top-left FIFO while the values of  $\mathbf{C}_{ij}^{k-1}$  are in the top-right FIFO. The complex results are stored in the bottom FIFOs.

multiplication of  $\mathbf{A}_{i1}\mathbf{B}_{1j}$  must be executed to generate all the elements of  $\mathbf{T}_{ij}$ .

- 2) To generate all the values of  $\mathbf{T}_{ij}$ , one must initialize and output the values of the BRAM into the corresponding FIFOs within our baseline *Complex Multiply-and-Accumulate* module depicted in Fig. 8. The initialization of the BRAM can be achieved by executing the outer products between the first column of the block  $\mathbf{A}_{i1}$  and the first row of block  $\mathbf{B}_{1j}$ . Similarly, the outputs can be generated by executing the outer products between the last column of block  $\mathbf{A}_{i1}$  and the last row of  $\mathbf{B}_{1j}$ .
- 3) One can take advantage of the sparsity of  $\mathbf{A}$  by utilizing a sparse matrix representation scheme. For instance, in computations where all the elements in the columns of  $\mathbf{A}_{i1}$  are zero, it is not necessary to read the corresponding row in the matrix  $\mathbf{B}_{1j}$  since the results of these multiplications are zero. The only exception to this situation is the second constraint mentioned previously.

To address the requirements mentioned above, we utilized a compressed sparse blocks (CSB) representation<sup>40</sup> with additional customized modifications. A schematic of this representation is shown in Fig. 10. To enable these parallelized calculations, Ref.<sup>40</sup> utilizes an integer array that contains the number of nonzero elements per block, where each block is represented using the compressed sparse row (CSR) representation. In this work, we also utilized an integer array containing the number of elements per block; however, we represent each block using a coordinate list format (COO) representation. In the COO representation, the input matrix can be represented in row- or column-major order; we chose the latter convention since this representation meets the requirements of our design. For computational efficiency, our implementation browses the block  $\mathbf{A}_{i1}$  one column at a time; thus, before matrix  $\mathbf{A}$  is sent from the CPU to the FPGA, it is first divided into blocks, and its CSB representation is generated. The matrices  $\mathbf{B}$  and  $\mathbf{C}$  are then sent to the FPGA as flat two dimensional arrays.

To accommodate the CSB representation of matrix  $\mathbf{A}$ , additional modifications of our

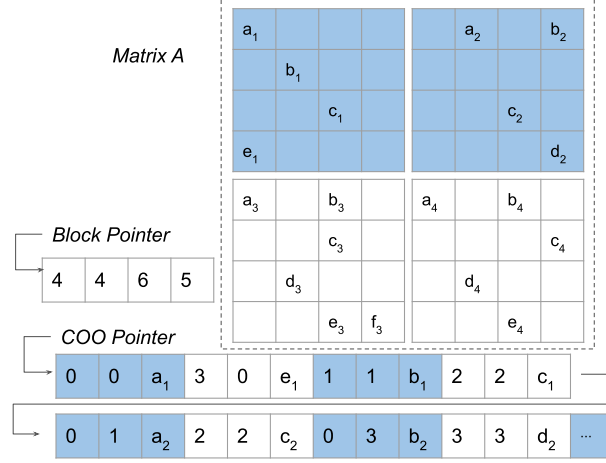


Figure 10: Schematic of the compressed sparse blocks (CSB) matrix representation used in this work. The input matrix is divided into four blocks of size  $4 \times 4$ . While the block pointer points to an array containing the number of nonzero elements per block, the coordinate list (COO) pointer points to an array containing the column index, row index, and the value of the nonzero elements in each block.

baseline implementation are required. These modifications only alter the *Read A Controller* and *Read B Controller* modules, with minor changes to the *Complex Multiply-and-Accumulate* module. Fig. 11 depicts our enhanced FPGA design where the *Read A Controller* now includes two additional input signals: the *Block Pointer* and the *COO Pointer*. Our enhanced design operates as follows:

- 1) To compute  $\mathbf{T}_{ij}$ , the *Read A Controller* signals the *Reader* to read the elements of block  $\mathbf{A}_{i1}$ . This operation makes use of the *Block Pointer* and *COO Pointer* signals.
- 2) The *Reader* places the elements of the COO array (the column index, row index, and real values of  $\mathbf{A}_{i1}$ ) into the FIFO labeled  $f_{ma}$ .
- 3) The *Read A Controller* reads the elements in the  $f_{ma}$  FIFO and signals the *Read B Controller* to read the next row (the column index in the COO array indicates the next row to read in the block  $\mathbf{B}_{1j}$ ). In addition, the *Read A Controller* places the elements of the columns of  $\mathbf{A}_{i1}$  into the  $a_{i,k}^r$  FIFO.
- 4) The *Read A Controller* notifies the *Complex Multiply-and-Accumulate* module (via the

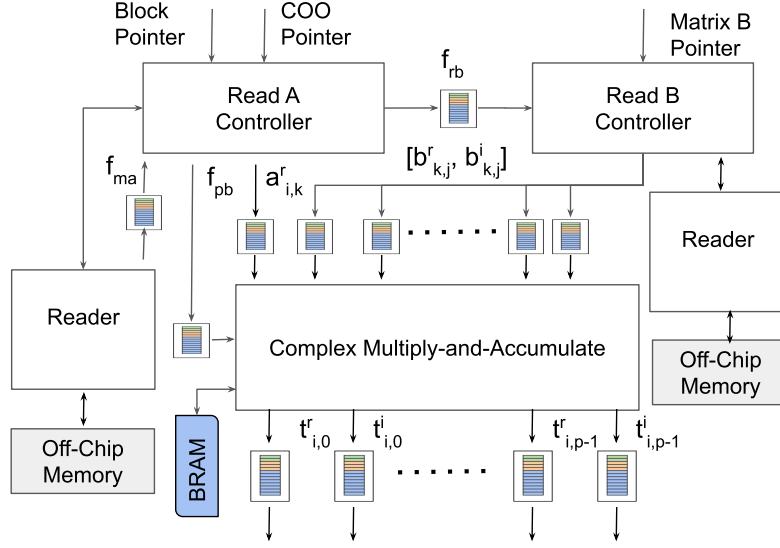


Figure 11: Hardware implementation of our optimized blocked complex-matrix multiplication module. Our implementation harnesses the *Block* and *COO* pointer to exploit the sparsity of  $\mathbf{A}_{i1}$ , and, as a result, dramatically decreases the complexity of computing  $\mathbf{A}_{i1}\mathbf{B}_{1j}$ .

$f_{pb}$  FIFO) when a new outer product between the columns of  $\mathbf{A}_{i1}$  and the rows of  $\mathbf{B}_{1j}$  has to be executed.

- 5) The *Complex Multiply-and-Accumulate* module reads the input  $b^r_{k,0}, b^i_{k,0}, \dots, b^r_{k,p-1}, b^i_{k,p-1}$  FIFOs as indicated in FIFO  $f_{pb}$ . In addition, this module reads the input FIFO  $a^r_{i,k}$ . These reading operations correspond to the next row in  $\mathbf{B}_{1j}$  and the next column-element in  $\mathbf{A}_{i1}$ , respectively.
- 6) Once these  $2p+1$  FIFOs are populated, the *Complex Multiply-and-Accumulate* module executes  $p$  complex multiply-and-accumulate operations and stores the results into the BRAM.
- 7) As in our baseline design, this process continues until all the outer products between the columns of  $\mathbf{A}_{i1}$  and the rows of  $\mathbf{B}_{1j}$  are completed. At the end,  $\mathbf{T}_{ij}$  is fully calculated.
- 8) Finally, the *Complex Accumulator*, shown in Fig. 9, takes  $\mathbf{T}_{ij}$  as input and computes  $\mathbf{C}^k_{ij}$  as described in our baseline design.

The FPGA design, as described previously, functions properly and efficiently in steady state, assuming that the pipelines do not have to be stalled. However, due to the sparsity of the input block  $\mathbf{A}_{i1}$ , we must account for stalls, which occur when a row of  $\mathbf{T}_{ij}$  is updated at cycle  $k$ , and later, when the same row has to be updated at cycle  $k + s$ . Because the *add* operation in the FPGA takes  $v$  cycles, these updates are allowed if  $s \geq v$ , otherwise we intentionally stall the pipeline for  $v - s$  cycles. The BRAM block shown at the bottom left of Fig. 11 is used to track when a row in  $\mathbf{T}_{ij}$  is updated. When row  $w$  of  $\mathbf{T}_{ij}$  gets updated, the *Complex Multiply-and-Accumulate* module writes the  $w$ th position of this BRAM with the value of a counter. If row  $w$  requires an update, the *Complex Multiply-and-Accumulate* module queries the BRAM at position  $w$  and determines whether the pipeline has to be stalled by comparing the current value of the counter with the value stored in the BRAM.

It is worth noting that when the elements in the  $f_{ma}$  FIFO are processed, the *Read A Controller* is able to signal to the *Read B Controller* which specific rows in block  $\mathbf{B}_{1j}$  to read. Each time that a row of  $\mathbf{B}_{1j}$  is skipped, significant savings in bandwidth, as well as in the number of floating point multiplications, are achieved. Thus, for every row skipped,  $4(2p)$  bytes are saved in I/O bandwidth, and  $p$  complex-valued multiplications and additions, are also avoided. As a result, by implementing all the FPGA acceleration strategies discussed previously, both the I/O as well as the complexity of computing  $\mathbf{T}_{ij}$  are significantly lowered.

## 6. Results and Discussion

### A. Single vs. Double Precision

To make our discussion of FPGA performance more concrete, we first present various metrics/benchmarks for calculating absorption spectra as a function of system size. Fig. 12 shows the absorption spectra obtained from RT-TDDFTB simulations carried out in single/double precision for several of the nanoribbons described in Section 3. The absorption spectrum for each nanoribbon was generated by propagating Eq. 1 in the presence of a Dirac

delta electric field impulse applied along three mutually orthogonal directions to compute the polarizability tensor. The resulting time-varying dipole moment was then Fourier transformed to give the absorption spectrum. Regardless of the nanoribbon size, Fig. 12 shows that the resulting spectra were extremely similar, independent of whether it was computed in single or double precision.

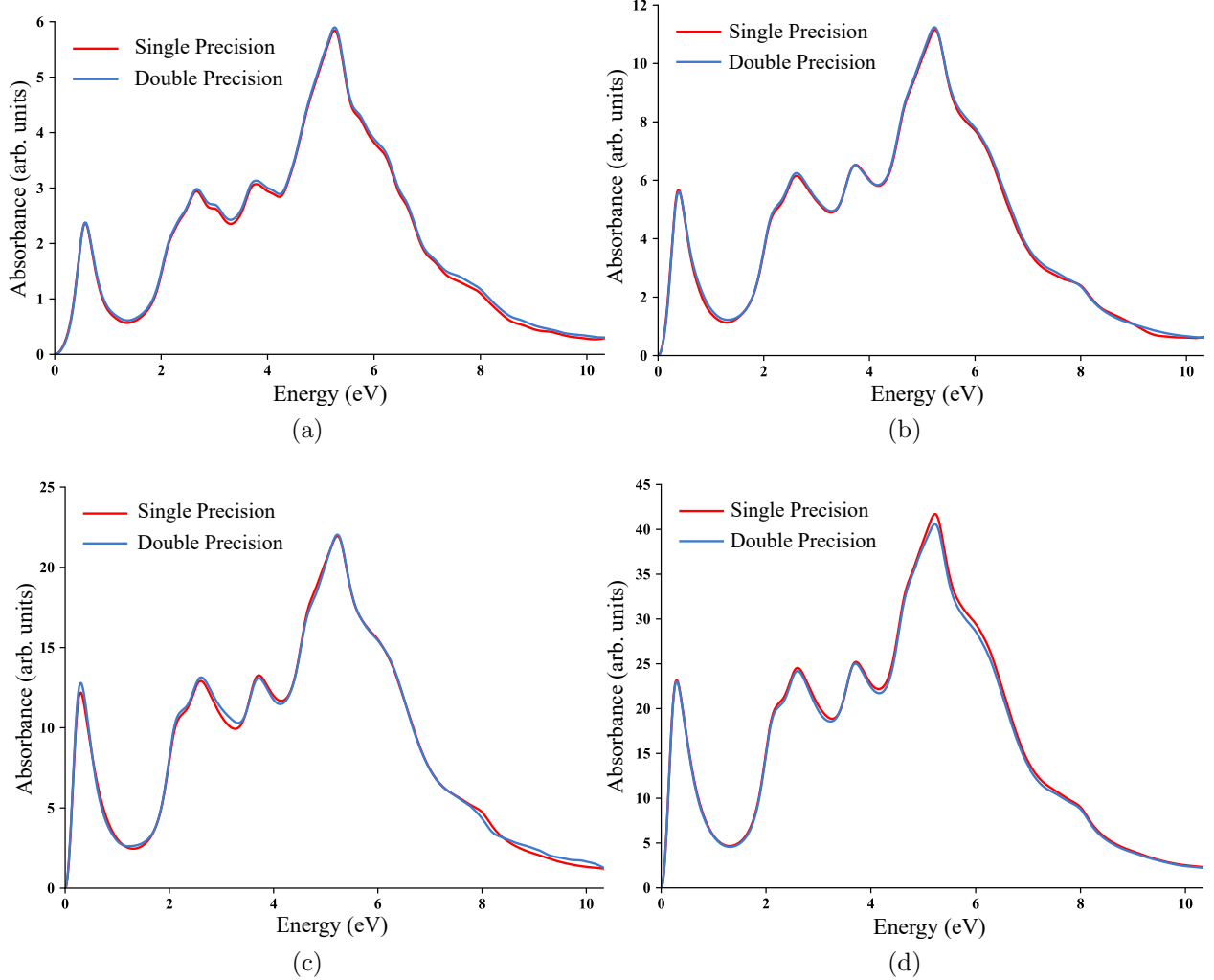


Figure 12: Absorption spectra of various carbon nanoribbons computed in single- and double-precision comprised of (a) 426, (b) 842, (c) 1,674, and (d) 3,338 atoms. In all cases, the absorption spectra computed in single precision is nearly indistinguishable from the double-precision spectra.

Table 1 gives a more quantitative comparison of numerical accuracy by calculating the

mean squared error (MSE) of the computed spectra according to the following expression

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2, \quad (6)$$

where  $y$  is the absorption spectrum of the nanoribbon computed in double precision and  $\hat{y}$  is the corresponding spectrum calculated in single precision. We obtained a maximum MSE of 4.2 (corresponding to the largest nanoribbon), with many of the smaller nanoribbons exhibiting much lower errors. We also carried out similar benchmark calculations on CPUs and GPUs and found that the MSEs between single/double precision for these hardware platforms were nearly identical to those reported in Table 1 for FPGAs (this is not surprising since the precision of arithmetic operations on CPUs,<sup>41</sup> GPUs,<sup>42</sup> and FPGAs<sup>43</sup> are required to be compliant with the IEEE-754 standard, which mandates the accuracy of single/double precision calculations on these hardware platforms). Nevertheless, these benchmark results are important since RT-TDDFTB calculations performed in single precision can significantly reduce FPGA I/O bandwidth and floating point operations.<sup>44</sup> Specifically, in FPGAs, the multiplication of two double or two single precision numbers requires 11 and 3 digital signal processors (DSPs), respectively.<sup>26,27</sup> As such, notwithstanding other hardware considerations, FPGAs can execute at least three times more multiplications per clock cycle when single precision arithmetic is used.

## B. Computational Speedup of FPGAs vs. GPUs and CPUs

In this section, we compare the computational efficiency of our RT-TDDFTB FPGA implementation against execution times obtained with the GPU and CPU. Because FPGAs are programmed/customized at the hardware level, we can fully take advantage of all the resources available on the FPGA by including all of the I/O channels and most of the block RAMs (nearly 70%). In particular, our RT-TDDFTB simulations were replicated such that four  $\mathbf{C}_{ij}$  blocks of size  $64 \times 64$  were computed in parallel, which allows  $512 = 4(64 \times 2)$  single-

Table 1: Mean squared errors (MSEs) between single vs double precision calculations of the absorption spectra for the various carbon nanoribbons computed with our FPGA-accelerated RT-TDDFTB approach.

Number of Atoms	Hamiltonian Matrix Size	MSE(%)
62	194	$\sim 0$
218	746	0.1
322	1,114	0.1
426	1,482	0.2
530	1,850	0.5
634	2,218	0.8
738	2,586	0.9
842	2,954	1.1
1,674	5,898	2.2
3,338	11,786	4.2

precision floating point operations per clock cycle. Table 2 provides a detailed accounting of the resources utilized by our FPGA implementation.

Table 2: Virtex-7 FPGA Utilization for Computing RT-TDDFTB Electron Dynamics

Resource	Available	Total Utilization (%)	Utilization per FPGA Engine (%)
Registers	2443K	45.11	11.27
LUTs	1221K	49.42	12.35
LUT RAM	344K	25.25	6.31
Block RAM	1.2K	74.27	18.56
FPUss	2.1K	55.56	13.89
Memory Channels	32	100	25

As mentioned in Section 2, to ensure a fair comparison of computational efficiency, our GPU-based RT-TDDFTB code used optimized cuSPARSE libraries to compute the  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\boldsymbol{\rho}]$  matrix in Compressed Sparse Column (CSC) format (similar to our FPGA design described in Section 5). However, we did not observe any gains in efficiency (GPUs are less efficient for sparse matrix operations, as discussed further in the paragraphs below); as such, we report GPU performance and energy metrics for calculations that only utilized the CUBLAS dense routines.<sup>28</sup> For our CPU calculations and comparisons, our RT-TDDFTB simulations

were executed on two and eight threads (i.e., eight CPU cores). Table 3 compares the total floating point operations (FLOPS) and FLOPS per clock cycle for the CPU, GPU, and FPGA on 5 representative carbon nanoribbon RT-TDDFTB simulations. These performance measurements correspond to steps 4 and 5 described previously in Section 2 (i.e., the steps offloaded to the co-processor for one iteration). The FLOPS for the GPU and CPU were obtained using the API<sup>45</sup> and LIKWID profiling tool,<sup>46</sup> respectively. For the FPGA, the number of FLOPS were directly counted. For each of the hardware platforms in Table 1, the FLOPS/cycle were calculated by first dividing the total FLOPS by the execution time (to get the FLOPS per second) and then dividing that number by the operating clock frequency of that hardware platform. From these results, we observe that the FLOPS are nearly the same for CPU and GPU, and we attribute deviations between these numbers to different routines and compiler optimizations for each platform. However, since the sparsity of  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\boldsymbol{\rho}]$  has been directly exploited on the FPGA at the hardware level, the resulting FLOPS have been reduced significantly. Most importantly, as the number of atoms in the simulation is increased, the FPGA executes more FLOPS/cycle. Consequently, in contrast to the CPU and GPU, the custom-designed pipelines on the FPGA operate in steady-state for a longer period and execute more arithmetic operations per clock cycle.

Table 3: Comparison of total floating point operations (FLOPS) and FLOPS per clock cycle for the CPU, GPU, and FPGA

Platform \ No. of Atoms	Total FLOPS ( $\times 10^9$ )					FLOPS/cycle				
	<b>218</b>	<b>426</b>	<b>842</b>	<b>1674</b>	<b>3338</b>	<b>218</b>	<b>426</b>	<b>842</b>	<b>1674</b>	<b>3338</b>
GPU	3.5	27.5	112.2	584.6	2860.1	88	93	96	95	105
CPU	3.7	29.3	135.4	596.2	3030.9	36	35	27	25	24
FPGA	1.1	6.2	15.6	52.6	203.4	100	111	117	126	134

In addition, we profiled our FPGA-enhanced code across all of our benchmark calculations, and the timings for each step are summarized in Table 4. From our profiling results, we observe that most of the execution time is spent on matrix multiplications (steps 2, 4, and 5). This is expected, since the complexity of matrix multiplication is  $\mathcal{O}(n^3)$  while the

computational complexity of steps 6.2 and 7 is  $\mathcal{O}(n^2)$ . In short, these timings justify the offloading of steps 4 and 5 to the co-processor since the acceleration of these steps will result in significant reductions in execution time and energy expenditures.

Table 4: Timings for each step in our FPGA-enhanced quantum dynamics simulations

Step	Timing (%)
1. Initialization	< 1
2. $S^{-1} \cdot \hat{H}[\hat{\rho}(t)]$	24
3. CPU to FPGA data transfer	2
4&5. Co-processor computations	60
6.1. FPGA to CPU data transfer	2
6.2. $\hat{\rho}(t) = \hat{\rho}_1(t) - \hat{\rho}_2(t)^T$	5
7. $\hat{\rho}$ and $\hat{H}$ update	6

Fig. 13 compares the computational speedup obtained with the FPGA, GPU, and CPU for nanoribbon systems containing 62 – 3,338 atoms. As is customary in hardware performance profiling, the computational speedup of each hardware platform is normalized by dividing its execution time by the timings of the CPU running two threads. For small RT-TDDFTB simulations on systems containing 62 – 530 atoms, the 8-thread CPU outperforms both the FPGA and GPU (with the GPU being slightly faster than the FPGA). The lower performance of the FPGA for these small system sizes can be attributed to two factors. First, since the FPGA relies on wide and deep pipelines to achieve high throughput, these pipelines are not able to reach a steady-state when the input matrices are small. Moreover, the latency of the off-chip memory (which is on the order of hundreds of cycles for the hardware used in this work<sup>24</sup>) results in further inefficiencies. These latencies severely underutilize the pipelines, and as a result, larger inputs are required before the pipelines achieve a steady-state. Second, for small calculations (i.e., those having only a few hundred atoms), the computational speedup obtained with GPUs and FPGAs does not compensate for the data-transfer communication time required to execute steps 4 and 5 with the co-processor (cf. Section 2). As such, significant computational speedup on GPUs and FPGAs can only be obtained for large chemical systems (which is the primary purpose of this work),

since the co-processor can carry out a significant portion of the calculation to compensate for the overhead associated with data transfer to and from the CPU. When the system size reaches 634 atoms, our FPGA implementation becomes more efficient than the GPU and is competitive with the 8-thread CPU. Finally, for large RT-TDDFTB simulations on systems containing over 842 atoms (where the sparsity is  $\sim 90\%$ , as shown in Fig. 4), our FPGA implementation outperforms both the GPU and CPU. Most importantly, as the number of atoms increases, the performance gap between the FPGA and the other competing hardware platforms increases as well (with the FPGA achieving a  $5\times$  speedup for the largest system).

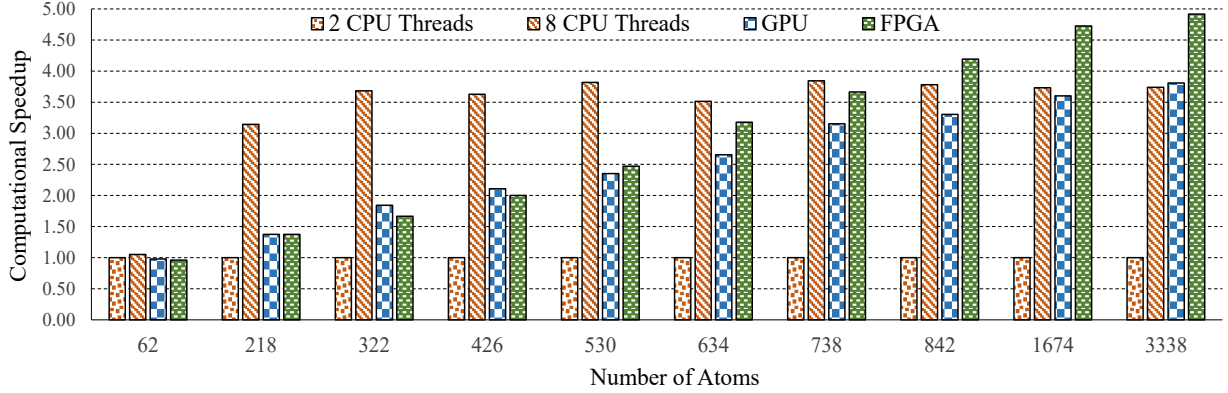


Figure 13: Comparison of computational speedup for the CPUs, GPUs (K40 architecture), and FPGAs (Virtex-7 architecture). For clarity, the speedup of each hardware platform is normalized by dividing its execution time by the timings of the CPU running two threads.

It is also worth mentioning that the computational performance of the GPU and CPU starts to saturate/plateau for large systems, whereas the performance of the FPGA continues to increase. Our FPGA implementation outperforms other platforms since it was specifically designed *at the hardware level* to take advantage of the sparsity of  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\rho}]$ , which effectively decreases the I/O and computational complexity of the problem (even more efficiently than GPUs). In particular, the performance of our FPGA implementation grows as a function of system size since the sparsity of  $\mathbf{S}^{-1} \cdot \hat{\mathbf{H}}[\hat{\rho}]$  increases with the number of atoms (cf. Fig. 4).

As demonstrated in our benchmark comparisons, it is important to emphasize that the multiplication of large, sparse matrices on GPUs is quite inefficient – a fact that has been

largely overlooked by the quantum chemistry/dynamics community. Specifically, GPUs are much better suited for dense matrix operations and achieve  $\sim 60\%$  of their theoretical peak performance, as measured in floating point operations per second, or FLOPS.<sup>47,48</sup> However, GPU performance significantly degrades as the sparsity of the input matrices increases (even when a sparse library is used), resulting in  $\sim 10\%$  of their theoretical peak performance.<sup>47,49,50</sup> GPUs suffer this significant drop in efficiency since they belong to a hardware classification known as single instruction multiple data (SIMD) architectures<sup>51</sup> – a class of computational architectures that can only execute *the same instruction over multiple streams of data*. In short, GPUs were specifically designed to deliver the highest throughput when the inputs are dense. In this scenario, GPUs (1) access contiguous chunks of data in off-chip memory via coalesced reads and writes, (2) provide a high off-chip memory bandwidth, (3) store efficiently small blocks of data in shared memories, and (4) execute a large number of floating point operations per clock cycle.<sup>52</sup> However, when the inputs of the matrices are sparse, GPUs encounter several difficulties that incur immense computational overhead, including: (1) storing the input matrices in a sparse matrix representation format, (2) accessing the data in an indirect fashion, since the metadata describing the input matrix has to be accessed before the data itself is read, (3) not having enough inputs (due to the sparsity of the input data) to fully saturate the floating point units, and (4) having a non-trivial distribution of equal work among the stream processors.<sup>53</sup> While FPGAs encounter the first, second, and fourth difficulties mentioned previously, their pipelines can be *customized* to take advantage of the granularity of the input data. Moreover, FPGAs can be adapted to provide fine-grained access to off-chip memory, flexible on-chip memory storage, and wide/deep pipelines to fully tackle the problem at hand.<sup>50,54</sup> As such, the use of FPGAs for these RT-TDDFTB electron dynamics applications shows significant performance gains (*even beyond modern GPUs*), particularly for large chemical/material systems.

### C. Energy Consumption of CPUs, GPUs, and FPGAs

In this section, we compare the energy consumption of each hardware platform, which is a practical and important metric (particularly for supercomputing centers mentioned in the Introduction) that has not been fully addressed by the quantum chemistry community. For the RT-TDDFTB electron dynamics performed on the GPU and FPGA, we measured the raw power by utilizing the NVIDIA management library and the Micron development kit,<sup>24,55</sup> respectively. For the GPU benchmarks, we utilized the correction scheme in Ref.<sup>56</sup> to give an accurate power estimation; in the CPU, the power was measured via the Likwid<sup>46</sup> suite. In our assessment of the FPGA and GPU platforms, the reported energy consumption does not include the energy consumed by the CPU since the majority of the computation (over 70%) was carried out on the co-processor (i.e., either the FPGA or GPU), and accounting for the power expenditures consumed by the CPU did not alter the observed trends.

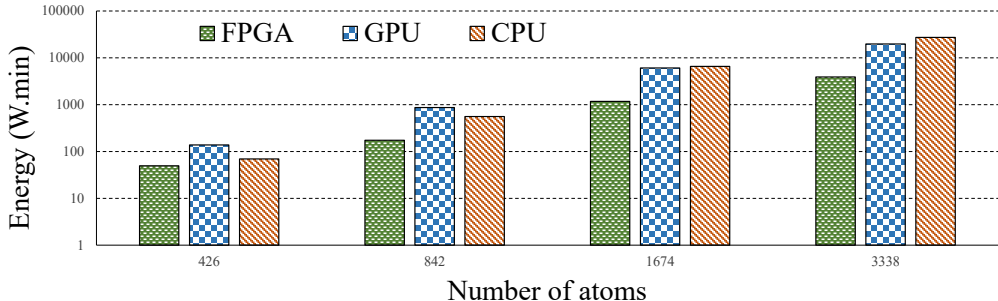


Figure 14: Comparison of energy consumption for FPGAs, GPUs, and CPUs (note the logarithmic scale on the vertical axis).

Fig. 14 compares the energy consumption in units of Watt-minutes for our four hardware platforms. The total energy was calculated by integrating the power as a function of time that each hardware platform consumes per calculation. In the GPU, the power usage was limited to 150 W. As shown in Fig. 14, the energy gap between the FPGA and GPU/CPU increases dramatically with the number of atoms in the system (note the logarithmic scale on the vertical axis). On average, the CPU and GPU consume 3.77 and 4.05 times more energy, respectively, than the FPGA, and this difference in energy efficiency is expected to further

increase with system size. Similar to the computational speedups described in Section 6.B., these massive calculations can be executed in an extremely energy-efficient manner since our FPGA implementation was specifically designed this way (by controlling individual electrical signals in the hardware). In general, CPUs and GPUs are relatively expensive to power with an energy efficiency of 10 MOPS/mW, whereas FPGAs are  $\sim 5$  times more cost effective with an efficiency of 50 MOPS/mW.<sup>8</sup>

## D. Performance on Other FPGA Hardware Architectures

To highlight the generality of our FPGA implementation, we also used additional optimization techniques (discussed at the end of Section 4.A.) to perform a few benchmark tests on newer FPGAs, namely the Kintex and Virtex UltraScale hardware architectures,<sup>57</sup> which we briefly discuss here. While our previous simulations were conducted on modern server-grade Virtex-7 FPGAs, we have migrated (synthesized, placed, and routed) our hardware design to a Virtex Ultrascale chip to forecast the gains in performance due to these newer hardware architectures. This migration is straightforward since the components used in our design are fully compatible with each other.<sup>26</sup> Specifically, (1) the Verilog modules used previously were directly implemented on this newer hardware, and (2) the Xilinx FPU and block RAMs were migrated effortlessly since these units are forward compatible. With these relatively easy alterations, our RT-TDDFTB electron dynamics could be executed at 266 MHz on the *VU9P* Virtex UltraScale chip (when our design is routed in this device, 52.5%, 50.8%, and 53.3% of the LUTs, block RAM, and ultra-block RAM resources are used, respectively). In short, because our design is quite general and can be placed/routed in a newer FPGA operating at 266 MHz (compared to the 167 MHz of our current FPGA), our implementation has the capability to run even faster, with additional performance gains of 62%, even beyond the computational speedup observed in Fig. 13.

## 7. Conclusion

In this work, we have presented the first application of field programmable gate arrays (FPGAs) for the fast and energy-efficient calculation of real-time electron dynamics in large chemical/material systems. Since FPGAs can be customized at the hardware level (even down to the level of specific electrical signals through user-defined gates), our implementation allows the simultaneous execution of several complex operations in an efficient manner, resulting in a truly optimized computational performance. Since FPGAs have not been previously used by the quantum dynamics community, we have provided a detailed description of our approach as a self-contained reference (with both hardware and programming details), followed with additional acceleration techniques tailored specifically to the efficient propagation of RT-TDDFTB electron dynamics. To thoroughly test and understand the performance of our new FPGA enhancements, we have examined a variety of performance benchmarks that include single vs. double precision tests, computational speedup comparisons against GPUs/CPUs, detailed energy consumption measurements, and an assessment of performance gains on other candidate FPGA hardware architectures.

By offloading the most intensive and repetitive calculations onto an FPGA, we show that the computational performance of our hardware implementation can even exceed that of optimized commercial mathematical libraries running on high-performance GPUs. In addition to this impressive computational speedup, we show that FPGAs are *immensely* energy-efficient and consume  $\sim 4$  times less energy than modern GPUs or CPUs. This latter metric is particularly promising since the power consumption of supercomputing centers (which incurs over \$1 million in power costs and causes deleterious climate change effects) is a contemporary topic that will need to be addressed soon, as exascale computing capabilities start to become more widespread and commonplace. Finally, it is worth mentioning that FPGA programming is at a similar developmental stage that GPUs started at 20 years ago; i.e., GPUs were first programmed in low-level machine languages but have now advanced to the stage where they are widely used to accelerate numerous quantum chemistry applications.

Likewise, FPGA performance has doubled in the last few years,<sup>58</sup> and the implementation techniques and performance metrics demonstrated in this work indicate that FPGAs could also play a similar and promising role in the acceleration (and energy-efficient calculation) of other types of quantum chemistry and materials science applications in the near future.

## Acknowledgement

J. M. R.-B., A. K., S. S. R. K. C. Y., W. N. and B. M. W. acknowledge support from the U.S. Department of Energy, Office of Science, Early Career Research Program under Award No. DE-SC0016269. M. B. O. acknowledges financial support from the Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT-FONCyT PICT-2017-0795). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

## References

- (1) Service, R. F. What I'll Take to Go Exascale. *Science* **2012**, *335*, 394–396.
- (2) U.S. Department of Energy, The Exascale Computing Project (ECP). <https://exascaleproject.org/exascale-computing-project>, (Accessed: 2019-10-01).
- (3) Lee, C. T.; Amaro, R. E. Exascale Computing: A New Dawn for Computational Biology. *Comput. Sci. Eng.* **2018**, *20*, 18–25.
- (4) Lefèvre, L.; Pierson, J.-M. Introduction to Special Issue on Sustainable Computing for Ultrascale Computing. *Sust. Comput. Journal* **2018**, *17*, 25–26.
- (5) Georgopoulos, K.; Mavroidis, I.; Lavagno, L.; Papaefstathiou, I.; Bakanov, K. *Hardware Accelerators in Data Centers*; Springer, 2019; pp 199–213.

- (6) Rigo, A. et al. Paving the Way Towards a Highly Energy-Efficient and Highly Integrated Compute Node for the Exascale Revolution: The ExaNoDe Approach. 2017 Euromicro Conference on Digital System Design (DSD). 2017; pp 486–493.
- (7) TOP500.org, The Top 500 List. <https://www.top500.org>, (Accessed: 2019-10-01).
- (8) Horowitz, M. Computing’s Energy Problem (And What We Can Do About It). International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). 2014; pp 10–14.
- (9) Sawyer, R. Calculating Total Power Requirements for Data Centers. *White Paper, American Power Conversion* **2004**, 562.
- (10) Shaw, D. E. et al. Anton, A Special-Purpose Machine for Molecular Dynamics Simulation. *Commun. ACM* **2008**, 51, 91–97.
- (11) Shaw, D. E. et al. Millisecond-Scale Molecular Dynamics Simulations on Anton. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. 2009; p 39.
- (12) Shaw, D. E.; Maragakis, P.; Lindorff-Larsen, K.; Piana, S.; Dror, R. O.; Eastwood, M. P.; Bank, J. A.; Jumper, J. M.; Salmon, J. K.; Shan, Y.; Wriggers, W. Atomic-Level Characterization of the Structural Dynamics of Proteins. *Science* **2010**, 330, 341–346.
- (13) Markets and Trends Incorporated, FPGA Market by Technology (SRAM, Antifuse, Flash), Node Size (Less than 28 nm, 28-90 nm, More than 90 nm), Configuration (High-End FPGA, Mid-Range FPGA, Low-End FPGA), Vertical (Telecommunications, Automotive), and Geography - Global Forecast to 2023. <https://www.marketsandmarkets.com/Market-Reports/fpga-market-194123367.html>, (Accessed: 2019-10-01).

- (14) Allec, S. I.; Sun, Y.; Sun, J.; Chang, C.-e. A.; Wong, B. M. Heterogeneous CPU+ GPU-Enabled Simulations for DFTB Molecular Dynamics of Large Chemical and Biological Systems. *J. Chem. Theory Comput.* **2019**, *15*, 2807–2815.
- (15) Ilawe, N. V.; Oviedo, M. B.; Wong, B. M. Real-Time Quantum Dynamics of Long-Range Electronic Excitation Transfer in Plasmonic Nanoantennas. *J. Chem. Theory Comput.* **2017**, *13*, 3442–3454.
- (16) Ilawe, N. V.; Oviedo, M. B.; Wong, B. M. Effect of Quantum Tunneling on the Efficiency of Excitation Energy Transfer in Plasmonic Nanoparticle Chain Waveguides. *J. Mater. Chem. C* **2018**, *6*, 5857–5864.
- (17) Oviedo, M. B.; Wong, B. M. Real-Time Quantum Dynamics Reveals Complex, Many-Body Interactions in Solvated Nanodroplets. *J. Chem. Theory Comput.* **2016**, *12*, 1862–1871.
- (18) Alvarez Barragan, A.; Ilawe, N. V.; Zhong, L.; Wong, B. M.; Mangolini, L. A Non-Thermal Plasma Route to Plasmonic TiN Nanoparticles. *J. Phys. Chem. C* **2017**, *121*, 2316–2322.
- (19) Oviedo, M. B.; Zarate, X.; Negre, C. F.; Schott, E.; Arratia-Pérez, R.; Sánchez, C. G. Quantum Dynamical Simulations as a Tool for Predicting Photoinjection Mechanisms in Dye-Sensitized TiO<sub>2</sub> Solar Cells. *J. Phys. Chem. Lett.* **2012**, *3*, 2548–2555.
- (20) Negre, C. F. A.; Fuertes, V. C.; Oviedo, M. B.; Oliva, F. Y.; Sánchez, C. G. Quantum Dynamics of Light-Induced Charge Injection in a Model Dye–Nanoparticle Complex. *J. Phys. Chem. C* **2012**, *116*, 14748–14753.
- (21) Negre, C. F. A.; Young, K. J.; Oviedo, M. B.; Allen, L. J.; Sánchez, C. G.; Jarzemska, K. N.; Benedict, J. B.; Crabtree, R. H.; Coppens, P.; Brudvig, G. W.; Batista, V. S. Photoelectrochemical Hole Injection Revealed in Polyoxotitanate Nanocrystals Func-

- tionalized with Organic Adsorbates. *J. Am. Chem. Soc.* **2014**, *136*, 16420–16429, PMID: 25337894.
- (22) Mukamel, S. *Principles of Nonlinear Optical Spectroscopy*, 1st ed.; Oxford University Press, New York, USA: 198 Madison Av, New York, USA, 1995; p 543.
- (23) Aradi, B.; Hourahine, B.; Frauenheim, T. DFTB+, A Sparse Matrix-Based Implementation of the DFTB Method. *J. Phys. Chem. A* **2007**, *111*, 5678–5684.
- (24) Micron Incorporated, Convey Wolverine Accelerator. <https://www.micron.com/>, (Accessed: 2017-09-04).
- (25) Mentor Incorporated, Mentor Incorporated. <https://www.mentor.com>, (Accessed: 2019-06-01).
- (26) Feist, T. Vivado Design Suite. *White Paper* **2012**, *5*.
- (27) Xilinx Incorporated, Xilinx DSP Slices (WP406). [https://www.xilinx.com/support/documentation/white\\_papers/wp406-DSP-Design-Productivity.pdf](https://www.xilinx.com/support/documentation/white_papers/wp406-DSP-Design-Productivity.pdf), (Accessed: 2018-09-04).
- (28) NVIDIA Incorporated, CUBLAS NVIDIA’s Dense Linear Algebra on GPUs. <https://docs.nvidia.com/cuda/cublas>, (Accessed: 2018-06-01).
- (29) Wong, B. M.; Ye, S. H.; O’Bryan, G. Reversible, Opto-Mechanically Induced Spin-Switching in a Nanoribbon-Spiropyran Hybrid Material. *Nanoscale* **2012**, *4*, 1321–1327.
- (30) Dou, Y.; Vassiliadis, S.; Kuzmanov, G. K.; Gaydadjiev, G. N. 64-bit Floating-Point FPGA Matrix Multiplication. Proceedings of the 13th International Symposium on Field-programmable Gate Arrays. 2005; pp 86–95.
- (31) Kumar, V. B.; Joshi, S.; Patkar, S. B.; Narayanan, H. FPGA Based High Performance Double-Precision Matrix Multiplication. *Int. J. Parallel Prog.* **2010**, *38*, 322–338.

- (32) Jovanović, Ž.; Milutinović, V. FPGA Accelerator for Floating-Point Matrix Multiplication. *IET Comput. Digit. Tec.* **2012**, *6*, 249–256.
- (33) Zhuo, L.; Prasanna, V. K. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems. *IEEE Trans. Parallel Distrib. Syst.* **2007**, *18*, 433–448.
- (34) Zhuo, L.; Prasanna, V. K. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. 18th International Parallel and Distributed Processing Symposium. 2004; p 92.
- (35) Ligon, W. B.; McMillan, S.; Monn, G.; Schoonover, K.; Stivers, F.; Underwood, K. D. A Re-Evaluation of the Practicality of Floating-Point Operations on FPGAs. Symposium on FPGAs for Custom Computing Machines. 1998; pp 206–215.
- (36) Bensaali, F.; Amira, A.; Sotudeh, R. Floating-Point Matrix Product on FPGA. International Conference on Computer Systems and Applications. 2007; pp 466–473.
- (37) Watkins, D. S. *Fundamentals of Matrix Computations*, 3rd ed.; John Wiley & Sons: 111 River St, Hoboken, NJ, USA, 2010.
- (38) Trefethen, L. N.; Bau III, D. *Numerical Linear Algebra*, 1st ed.; Siam: 3600 Market Street, 6th Floor, Philadelphia, PA, 19104, USA, 1997.
- (39) Gilles, K. The Semantics of a Simple Language for Parallel Programming. *Inf. Process. Lett.* **1974**, *74*, 471–475.
- (40) Buluç, A.; Fineman, J. T.; Frigo, M.; Gilbert, J. R.; Leiserson, C. E. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures. 2009; pp 233–244.

- (41) Goldberg, D. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.* **1991**, *23*, 5–48.
- (42) Whitehead, N.; Fit-Florea, A. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *rn (A + B)* **2011**, *21*, 18749–19424.
- (43) Xilinx Incorporated, Xilinx DSP Slices (WP406). [https://www.xilinx.com/support/documentation/ip\\_documentation/floating\\_point/v7\\_1/pg060-floating-point.pdf](https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf), (Accessed: 2018-09-04).
- (44) Pokhilko, P.; Epifanovsky, E.; Krylov, A. I. Double Precision is Not Needed for Many-Body Calculations: Emergent Conventional Wisdom. *J. Chem. Theory Comput.* **2018**, *14*, 4088–4096.
- (45) NVIDIA Incorporated, CUDA Toolkit Documentation. <https://docs.nvidia.com/cuda>, (Accessed: 2018-09-04).
- (46) Treibig, J.; Hager, G.; Wellein, G. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. 39th IEEE International Conference on Parallel Processing Workshops. San Diego, CA, USA, 2010; pp 207–216.
- (47) Danalis, A.; Marin, G.; McCurdy, C.; Meredith, J. S.; Roth, P. C.; Spafford, K.; Tipparaju, V.; Vetter, J. S. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. 2010; pp 63–74.
- (48) Yang, C.; Buluç, A.; Owens, J. D. Design Principles for Sparse Matrix Multiplication on the GPU. European Conference on Parallel Processing. 2018; pp 672–687.
- (49) Zhang, Y.; Shalabi, Y. H.; Jain, R.; Nagar, K. K.; Bakos, J. D. FPGA vs. GPU for Sparse Matrix Vector Multiply. 2009 International Conference on Field-Programmable Technology. 2009; pp 255–262.

- (50) Halstead, R. J.; Villarreal, J.; Najjar, W. Exploring Irregular Memory Accesses on FPGAs. Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms. 2011; pp 31–34.
- (51) Hennessy, J. L.; Patterson, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed.; Morgan Kaufmann: 225 Wyman St, Waltham, MA, USA, 2012.
- (52) Kirk, D. B.; Hwu, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd ed.; Morgan kaufmann: 225 Wyman Street, Waltham, MA, 02451, USA, 2013.
- (53) Liu, W.; Vinter, B. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. Proceedings of the 29th ACM on International Conference on Supercomputing. 2015; pp 339–350.
- (54) Sirowy, S.; Forin, A. *Where's the Beef? Why FPGAs are so Fast*; Microsoft Research, Technical Report, MSR-TR-2008-130, 2008.
- (55) Anderson, J. H.; Najm, F. N. Power Estimation Techniques for FPGAs. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2004**, *12*, 1015–1027.
- (56) Burtscher, M.; Zecena, I.; Zong, Z. Measuring GPU Power with the K20 Built-In Sensor. Proceedings of Workshop on General Purpose Processing Using GPUs. 2014; pp 28–36.
- (57) Xilinx Incorporated, UltraScale Architecture and Product Data Sheet: Overview (DS890). [https://www.xilinx.com/support/documentation/data\\_sheets/ds890-ultrascale-overview.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf), (Accessed: 2019-09-01).
- (58) Shannon, L.; Cojocaru, V.; Dao, C. N.; Leong, P. H. Technology Scaling in FPGAs: Trends in Applications and Architectures. 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines. 2015; pp 1–8.